K-means Algorithm

Federico Nocentini
E-mail address
federico.nocentini@stud.unifi.it

Corso Vignoli
E-mail address
corso.vignoli@stud.unifi.it

Abstract

K-Means is a parametric clustering technique with an embarassingly parallel structure and for this reason it's suitable for parallel computing. In this work, an OpenMP and a CUDA implementation will be presented and the execution times of each version will be compared. A particular focus will be given to the speedup obtained with the parallel versions for datasets of increasing dimension.

1. Introduction

K-means clustering is one of the simplest and most popular unsupervised machine learning algorithms. A cluster refers to a collection of data points aggregated together because of certain similarities. We define a target number K, which refers to the number of centroids we need in the dataset. A centroid is the imaginary or real location that represents the center of the cluster. The 'means' in K-means stands for the average of the data : the goal is finding the centroid. K-means, as it is structured, works only in Euclidean spaces. This is due to a fundamental property of Euclidean spaces: the average of points always exists and is a point in the space. To process the learning data, the K-means algorithm starts with a first group of K randomly selected centroids, which are used as beginning points for every cluster, and then performs iterative calculations to optimize the positions of the centroids.

1.1. Formal Definition

Given an initial set of observations in a d-dimensional Euclidean space:

$$x_1, x_2, \ldots, x_N;$$

Given a set of k random centroids:

$$c_1^{(1)}, c_2^{(1)}, \ldots, c_K^{(1)};$$

the algorithm proceeds by alternating between two steps:

Assignment step: Assign each observation to the cluster with the nearest centroid, the one with the least squared Euclidean distance.

$$C_i^{(t)} = \left\{ x_p : \|x_p - c_i^{(t)}\|^2 \le \|x_p - c_j^{(t)}\|^2, \forall j : 1 \le j \le K \right\}$$

Update Step: Recalculate centroids for observations assigned to each cluster.

$$c_i^{(t+1)} = \frac{1}{|C_i^{(t)}|} \sum_{x_j \in C_i^{(t)}} x_j$$

The algorithm converges when assignments and centroids no longer change.

2. Proposed Approach

In this paper we propose 3 different implementations of the algorithm:

1. A parallel version with CUDA
2. A parallel version with OpenMP
3. A sequential version with Python

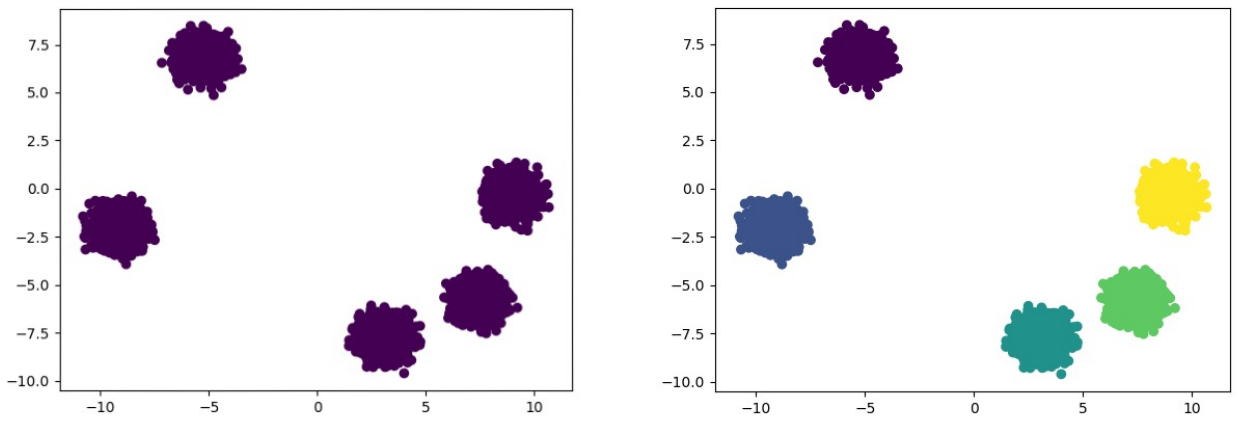

Figure 1. Example of a dataset with 10000 points respectively not clustered and clustered

We choose to build implementations of K-means that work in 2-dimensional Euclidean spaces and the centroids are initialized by taking K points at random from the N observations.

2.1. Python

This sequential implementation is a simple translation of the principle behind the K-means in Python. The *Algorithm 1* shows the pseudocode of the core of the algorithm:

Algorithm 1 K-means Core

```
function K-MEANS($points, centroids, assgn$)
    while iterationIndex $< MAX\_ITERATIONS$ do
        $assgn \leftarrow ASSGN(points, centroids, assgn)$
        $centroids \leftarrow UPDATE(points, assign)$
```

The $MAX\_ITERATIONS$ constant represents the number of iterations of the algorithm. The *Algorithm 2* shows the first step of a K-means iteration, the assignment of each points to the cluster with the nearest centroid:

Algorithm 2 K-means assignment

```
function ASSGN($points, centroids, assgn$)
    for each point $p$ in $points$ do
        $dist \leftarrow INF$
        for each centroid $c$ in $centroids$ do
            $d \leftarrow L_2DISTANCE(p, c)$
            if $d < dist$ then
                $dist \leftarrow d$
                $assgn[p] \leftarrow c$
    return $assgn$
```

The *Algorithm 3* shows the second step of a K-means iteration, the clusters centroids update:

Algorithm 3 K-means update

```
function UPDATE($points, assgn$)
    $centroidSum \leftarrow 0$
    $clusterSize \leftarrow 0$
    for each point $p$ in $points$ do
        $clusterId \leftarrow assgn[p]$
        $clusterSize[clusterId]$ += 1
        $centroidSum[clusterId]$ += $p$
    return $centroidSum/clusterSize$
```

2.2. OpenMp

The OpenMP library lets us transform the sequential version into a parallel one with several **pragma** directive. The core of the implementation is the same as the sequential one, the only thing that changes is the *#pragma omp parallel* directive in *Algorithm 4* which allows us to compute our code in parallel on the CPU. Unlike the sequential implementaion, the sum of the points assigned to each cluster is computed in the assignment step. So, for this reason, we need to call that section "critical".

Algorithm 4 K-means OpenMP assignment/update

```
function ASSGN_UPDATE($points, clusters$)
    #pragma omp for schedule(static)
    for each point $p$ in $points$ do
        $dist \leftarrow INF$
        $min\_index \leftarrow 0$
        for each cluster $c$ in $clusters$ do
            $d \leftarrow L_2DISTANCE(p, c)$
            if $d < dist$ then
                $dist \leftarrow d$
                $min\_index \leftarrow c$
        $p.set\_cluster\_id(min\_index)$
        #pragma omp critical
        $c[min\_index].add\_point(p)$
```

2.3. CUDA

Implementing the algorithm with CUDA lets us take advantage of the great number of cores of GPUs. Indeed, the processing of each point can be assigned to a different thread. In our implementation, the N points to be clustered are stored in 2 arrays of one dimension, which contain the features of each point, respectively:

$$p_x = [x_1, x_2, \dots, x_N]\,, p_y = [y_1, y_2, \dots, y_N]$$

The same data organization is applied to centroids:

$$c_x = [x_1, x_2, \dots, x_K]\,, c_y = [y_1, y_2, \dots, y_K]$$

The *Algorithm 5* shows the pseudocode of the core of the algorithm.
The *Algorithm 6* shows the kernel that assigns each point to the nearest centroid.
The *Algorithm 7* shows the kernel that computes the sum of the respective coordinates of all points assigned to each cluster. The actual centroids calculation is done in the core, dividing the sum by the number of assigned points.

Algorithm 5 K-means Core CUDA

```
function K-MEANS($p_x, p_y, c_x, c_y, assgn, c\_size$)
    while iterationIndex $< MAX\_ITERATIONS$ do
        C_ASSGN($p_x, p_y, c_x, c_y, assgn$)
        $sum_x \leftarrow 0$
        $sum_y \leftarrow 0$
        $c\_size \leftarrow 0$
        C_UPDATE($p_x, p_y, sum_x, sum_y, assgn, c\_size$)
        $c_x \leftarrow sum_x / c\_size$
        $c_y \leftarrow sum_y / c\_size$
```

Algorithm 6 K-means Assignment CUDA

```
function C_ASSGN($p_x, p_y, c_x, c_y, assgn$)
    $idx \leftarrow blockIdx.x * blockDim.x + threadIdx.x$
    if $idx > N$ then
        return
    $dist \leftarrow INF$
    $closest\_centroid \leftarrow 0$
    $c \leftarrow 0$
    while $c < K$ do
        $d \leftarrow DISTANCE(p_x[idx], p_y[idx], c_x[c], c_y[c])$
        if $d < dist$ then
            $dist \leftarrow d$
            $closest\_centroid \leftarrow c$
        $c \leftarrow c + 1$
    $assgn[idx] \leftarrow c$
```

Algorithm 7 K-means Update CUDA

```
function C_UPDATE($p_x, p_y, sum_x, sum_y, assgn, c\_size$)
    $idx \leftarrow blockIdx.x * blockDim.x + threadIdx.x$
    if $idx > N$ then
        return
    $cluster\_id \leftarrow assgn[idx]$
    $sum_x[cluster\_id] \leftarrow sum_x[cluster\_id] + p_x[idx]$
    $sum_y[cluster\_id] \leftarrow sum_y[cluster\_id] + p_y[idx]$
    $c\_size[cluster\_id] \leftarrow c\_size[cluster\_id] + 1$
```

3. Experimental Results

The metric used to compare the performances of the sequential algorithm with the OpenMP and the CUDA versions is the **speedup**, computed as:

$$S = \frac{t_S}{t_P}$$

where t_S and t_P are respectively the execution time of the sequential and the parallel implementation. The datasets used to evaluate the different implementations have been generated with the

$make\_blob()$ function of *sklearn.datasets* [1]. They are gaussian distributions with 5 centers and standard deviation equal to 0.5 and are composed by respectively 100, 1000, 10000, 100000 and 1000000 2D points.
The $MAX\_ITERATIONS$ constant has been set to 20 because it has been estimated empirically that 20 iterations are enough to make all the centroids converge.
The tests have been executed on a machine with:

- OS: Ubuntu 18.04 LTS
- CPU: Intel® Core™ i7-10710U, 6 Cores/12 Threads
- RAM: 16 GB DDR4
- GPU: GeForce GTX 1650 Mobile / Max-Q 4GB with CUDA 10.1

To make the results more reliable and representative, each execution time has been obtained as the average of the times measured running each test 5 times for the sequential and the OpenMP versions and 15 times for each CUDA implementation.

3.1. OpenMP

To evaluate the performances of the OpenMP implementation, it has been executed on each dataset with an increasing number of threads. The results for the 100, 1000, 10000, 100000 and 1000000 dataset are shown respectively in *Table 1*, *Table 2*, *Table 4*, *Table 6* and *Table 5*.

Dim	100	
Sequential	0.008295 s	
Thread	**Time**	**Speedup**
2	0.000591 s	14.04
3	0.000555 s	14.94
4	**0.000532** s	**15.59**
5	0.000600 s	13.82
6	0.000622 s	13.33
7	0.000664	12.49
8	0.000714	11.61
9	0.000716	11.58
10	0.000749	11.07
11	0.000829	10.00
12	0.000892	9.29

Table 1. Speedup for 100 points varying the number of threads with OpenMP (best result in bold)

Dim	1000	
Sequential	0.097896 s	
Thread	**Time**	**Speedup**
2	0.007247 s	13.50
3	0.007157 s	13.67
4	0.007065 s	13.85
5	0.006987 s	14.01
6	0.006856 s	14.27
7	0.006734 s	14.53
8	0.006732 s	14.54
9	0.006698 s	14.61
10	0.006623 s	14.76
11	0.006545 s	14.95
12	**0.006432 s**	**15.22**

Table 2. Speedup for 1000 points varying the number of threads with OpenMP (best result in bold)

For the 100 points dataset (*table 1*), it's curious to see that the speedup decreases for more than 4 threads. This shows how, for such a low number of points, the overhead of the threads outweighs the gain from using them. As expected, this phenomenon disappears in the datasets with more points and the use of more threads lowers the execution time (and consequently increases the speedup). The results show how OpenMP lets us reach a speedup equal to at least **15** at the expense of several pragma directive, so OpenMP proves to have an excellent speedup / development cost ratio.

Dim	10000	
Sequential	1.04959 s	
Thread	**Time**	**Speedup**
2	0.07651 s	13.71
3	0.07313 s	14.35
4	0.07302 s	14.37
5	0.07292 s	14.39
6	0.07232 s	14.51
7	0.07192 s	14.59
8	0.07134 s	14.71
9	0.07098 s	14.78
10	0.07002 s	14.98
11	0.06958 s	15.08
12	**0.06903 s**	**15.20**

Table 4. Speedup for 10000 points varying the number of threads with OpenMP (best result in bold)

Dim	Sequential	OpenMP	OpenMP Speedup	CUDA	CUDA Speedup
100	0.008295 s	0.000532 s	15.59	0.000958 s	8.65
1000	0.097896 s	0.006432 s	15.22	0.003188 s	31.30
10000	1.04959 s	0.06903 s	15.20	0.005306 s	197.1
100000	13.24 s	0.871 s	15.20	0.067217 s	196.3
1000000	102.40 s	6.71 s	15.26	0.5131 s	200.7

Table 3. Global comparison between sequential, OpenMP and CUDA best results varying dataset dimension

Dim	100000	
Sequential	13.24 s	
Thread	**Time**	**Speedup**
2	0.981 s	13.49
3	0.972 s	13.62
4	0.965 s	13.72
5	0.962 s	13.76
6	0.950 s	13.93
7	0.941 s	14.08
8	0.934 s	14.17
9	0.921 s	14.37
10	0.917 s	14.43
11	0.905 s	14.62
12	**0.871 s**	**15.20**

Table 5. Speedup for 100000 points varying the number of threads with OpenMP (best result in bold)

Dim	1000000	
Sequential	102.40 s	
Thread	**Time**	**Speedup**
2	7.57 s	13.52
3	7.32 s	13.98
4	7.27 s	14.08
5	7.17 s	14.28
6	7.06 s	14.50
7	7.03 s	14.56
8	6.99 s	14.64
9	6.90 s	14.84
10	6.97 s	14.90
11	6.80 s	15.05
12	**6.71 s**	**15.26**

Table 6. Speedup for 1000000 points varying the number of threads with OpenMP (best result in bold)

3.2. CUDA

To evaluate our CUDA implementation, we test the block dimension with fixed dataset dimension, 10000 points.

Block Dim	CUDA
32	0.007066 s
64	**0.005472 s**
128	0.006636 s
256	0.009060 s
512	0.009876 s
1024	0.011584s

Table 7. Execution time for 10000 points dataset while changing the block dimension (best result in bold)

To evaluate the performances of the CUDA implementation, it has been executed on each dataset like the OpenMP implementation with block dimension fixed at **64**. The results are shows in *table 8*:

Dim	Sequential	CUDA	Speedup
100	0.008295 s	0.000958 s	8.65
1000	0.097896 s	0.003188 s	31.30
10000	1.04959 s	0.005306 s	197.1
100000	13.24 s	0.067217 s	196.3
1000000	102.40 s	0.5131 s	200.7

Table 8. Execution time and speedup obtained with sequential and CUDA implementations for increasing dataset dimension

3.3. Comparison

As a final result, a global comparison has been conducted and therefore only the best results for each dataset dimension and each implementation have been considered. In *table 3* we can see that the CUDA algorithm abundantly **outperforms** both the sequential and the OpenMP ones, at the expense of a more complicated implementation. However OpenMP lets us to achieve a noticable speedup with just several directive.

4. Conclusions

In this work, the K-means clustering algorithm was presented and it was shown how its embarassingly parallel structure makes it suitable for parallel computing. A parallel implementation with

OpenMP was developed by adding just a directive and it allows to obtain a speedup equal to more than 15. Then, a CUDA implementation was presented and, with its 200 speedup, it showed how the use of GPUs makes K-means applicable to datasets intractable with a CPU.

References

[1] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, *et al.*, “" scikit-learn: Machine learning in python," journal of machine learning research, vol. 12, p,” 2011.

[2] M. Ahmed, R. Seraj, and S. M. S. Islam, “The k-means algorithm: A comprehensive survey and performance evaluation,” *Electronics*, vol. 9, no. 8, p. 1295, 2020.

[3] Y. Zhang, Z. Xiong, J. Mao, and L. Ou, “The study of parallel k-means algorithm,” in *2006 6th World Congress on Intelligent Control and Automation*, vol. 2, pp. 5868–5871, IEEE, 2006.

[4] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[5] B. Hong-Tao, H. Li-Li, O. Dan-Tong, L. Zhan-Shan, and L. He, “K-means on commodity gpus with cuda,” in *2009 WRI World Congress on Computer Science and Information Engineering*, vol. 3, pp. 651–655, IEEE, 2009.

[6] C. Zeller, “Cuda c/c++ basics,” *NVIDIA Coporation*, 2011.